Sorting and Searching

In the introductory example, we used a linear search to find a particular item in an array. Now let's start utilising what you learnt in CSCI103 – but with some revision of the material as well. Let's start by looking at three simple ways of sorting data.

1. Three Simple Sorts

Suppose we have an array of n integer values stored in the array X. Let's assume we want this array ordered from smallest to largest value.

1.1 Selection Sort

We start by scanning the array to find the smallest value. If this value is not already at the position X[0], swap it with the contents of X[0]. We now have the smallest value in the correct position. Then the remainder of the array, i.e. X[1..n-1] is searched for next smallest, which is swapped for X[1]. Proceed through the array.

After each pass through the array, one item is guaranteed to be in its correct position. So, after n passes, each through smaller portions of the array, we can guarantee the entire array is sorted. Here is an implementation.

```
for (i=0;i<n-1;i++)
{
     smallest = i;              // location of smallest so far
     for (j=i+1;j<n;j++)
          if (X[smallest] > X[j])
               smallest = j;    // new location of smallest
     if (smallest != i)         // swap if not already in position
     {
          temp = X[i];
          X[i] = X[smallest];
          X[smallest] = temp;
     }
}
```

When we looked at the linear search earlier, we indicated that the number of checks needed indicated the efficiency of the method. Note here how there are two loops involved, one nested inside the other. The outer loop is performed n-1 times. The inner loop is performed a lesser number of times for each step of the outer loop: n-1 the first time (i=0), n-2 the next, n-3, n-4, . . . down to once when i=n-2. So the comparison inside the inner loop is performed a total of

$$(n-1) + (n-2) + (n-3) + \ldots + 1 = n(n-1)/2 \text{ times.}$$

We'll see in a later section how we use this value to compare efficiency. The number of exchanges needed is less than n.

1.2 Bubble Sort

This involves the same number of passes through the array as Selection Sort, but we move more values. On each pass, consecutive values are compared and swapped if in the wrong order. So the big values will 'bubble' up towards the end. (We are sorting into increasing order again.) In fact, after each pass, one extra value will be guaranteed to be in its correct position towards the end of the array.

So,

```
for (i=n-1;i>0;i--)
{
     for (j=0;j<i;j++)
          if (X[j] > X[j+1])
               swap values j and j+1
}
```

This approach involves the same number of comparisons as Selection Sort but we do a lot more swaps. Note that, at each pass through the outer loop, the variable i is the last value in the array that is yet to

be sorted. Let's modify the outer loop to a while loop, changing the variable name to reflect its meaning.

```
LastUnsorted = n-1;
while (LastUnsorted > 0)
{
     for (j=0;j<LastUnsorted;j++)
          if (X[j] > X[j+1])
               swap values j and j+1
     LastUnsorted--;
}
```

We can further improve on this. If, during the inner loop, no swaps were needed after, say, `LastSwapIndex`, then the remaining entries from `X[LastSwapIndex]` to `X[LastUnsorted-1]` and hence to `X[n-1]` must already be ordered.

So,

```
LastUnsorted = n-1;
while (LastUnsorted > 0)
{
     LastSwapIndex = 0;
     for (j=0;j<LastUnsorted;j++)
     {
          if (X[j] > X[j+1])
          {
               swap X[j] and X[j+1]
               LastSwapIndex = j;
          }
     }
     LastUnsorted = LastSwapIndex;
}
```

This means Bubble Sort will (most likely) require a smaller number of comparisons than Selection Sort. But how does it perform as n increases? We'll see later.

1.3 Insertion Sort

This algorithm is essentially how a person might sort a set of values when given the items to sort one at a time. We start with one value (sorted). We then look at the next item to be added to the order, finding where it should be placed in relation to the number(s) that are already sorted. Let us again assume we want increasing order.

So, for each item to be added to the sorted set there are two steps:

- Find where the item goes.
- Put it there.

Given the array `X[0..i-1]` of already sorted items, how do we find where the value `Item` fits?

```
Pos = 0;
while (Pos < i && Item >= X[Pos])
     Pos++;
```

Upon completion `Pos` has the value of the index at which the value `Item` must go, either at `Pos=i` or before `X[Pos]`.

If the former, then

```
X[Pos] = Item;
i++;
```

whereas, in the latter case, the array elements must be moved to accommodate the extra value

```
for (j=i-1;j>=Pos;j--);
      X[j+1] = X[j];
```

and then

```
X[Pos] = Item;
i++;
```

We can actually combine the two steps by searching for the insertion point from the upper end, moving the array elements up as we go. Finally we have

```
Pos = i-1;
while (Pos >= 0 && Item < X[Pos])
{
     X[Pos+1] = X[Pos];
     Pos--;
}
X[Pos+1] = Item;
i++;
```

This algorithm can be extended to sorting an already occupied but unsorted array `X`, by starting with a one-long sorted subarray containing `X[0]`. Now we find where `Item=X[1]` fits in this sorted array. Then where `Item=X[2]` fits into the now sorted array of length 2. And so on. Thus, at any stage in the process, all entries so far inserted are in the correct order.

As for the number of comparisons required, on average, the value being inserted will be placed in the middle of the values already sorted. Thus, Insertion Sort requires half the comparisons of Selection Sort. So `n(n-1)/4` comparisons will be needed – on average.

Thus we now have three sorting methods:

Selection
Bubble
Insertion

Which is the best? How can we compare their efficiency?

2. Efficiency and Proof

In the last section we talked about guarantees. That is, we had to be sure that, if we followed the procedure described, by the completion of the process the data would be sorted. Whenever an algorithm is designed, we must be certain that the algorithm works. Just because some test runs of a particular implementation produces the correct result is not a **proof of correctness**. The advantage of having a knowledge of **standard algorithms** – tried-and-tested methods – is that we can be assured that the procedure will work. This is the first criterion for a good algorithm.

The second criterion for judging an algorithm is **efficiency**. We want methods that perform well on our data. In particular, the efficiency of algorithms that are designed for manipulating data sets of a certain size can be described by how they perform as the size of the data set increases. When it comes to sorting algorithms, the number of comparisons which have to be made is a good indication of efficiency (although the number of exchanges of data items is also of importance). Let's look at Selection Sort first.

We stated above that Selection Sort involves $n(n-1)/2$ comparisons. For those with a mathematical mind, we say that such an expression is of the **order** of n^2, as the expression $0.5n^2-0.5n$ behaves like a constant times n^2 when n is large. (In actuality, we are saying that the number of comparisons is **bounded above** by a constant times n^2 for values of n beyond some particular integer.) The linear term is overpowered by the square, and is ignored. We write this as $O(n^2)$ – called **big-O notation**. In fact all three of the algorithms above are $O(n^2)$ sorts. But that doesn't mean that they are all equal. They differ in the constant multiplier – and whether this is an average or a constant performance. It does, however, mean that doubling the data set size will quadruple the time it takes to do the sort – on average. So how do we decide which of the above three is best?

We look at best and worst case performance. That is, are there initial arrangements of the data that make the algorithms perform better or worse than $O(n^2)$?

For Selection Sort, it doesn't matter. Because the loops are of fixed lengths, the sort will always take exactly the same time. Obviously, if the data is already sorted, there will be no exchanges, but the comparisons will all take place.

For Bubble Sort, an already sorted data set will involve one pass through the data, with no exchanges taking place. The algorithm will detect that no further passes are necessary and so only *n-1* comparisons are needed – *O(n)*. If the data is completely reversed, the full number of comparisons will be needed, as each pass through the data will move one item from the front of the set to the back with no change in the relative positions of other entries. Thus the only thing we can say is that the worst case is $O(n^2)$, and that on average less than $n(n-1)/2$ comparisons will be needed.

For Insertion Sort, best case is sorted, where we need only compare each new entry to the end of the ordered part of the data to indicate that the new item goes where it already is. The worst case is completely reversed data, where we have to compare each new entry to all already-sorted data. For the average case, we previously stated that about half the already sorted values would need to be checked, so $n(n-1)/4$ comparisons would be needed.

Thus, Selection Sort is worst, with Bubble and Insertion Sort somewhat similar. The advantage of Insertion Sort is that extra data items can be inserted into an already sorted set efficiently.

Can we improve on $O(n^2)$?

3. Binary Search

We indicated earlier that the searching of a database could be improved by having the data sorted by the key used to search. This would enable us to shorten the search for key values not present in the data. It also enables us to find a more efficient way of locating key values that are present in the data. Now that we have the concept of order, we can say that linear search is an *O(n)* process, as the number of comparisons needed behaves like a constant times the size of the set.

An improved method of locating a value in a set of values is **binary search**.
Suppose we have a sorted set of n values in an array `X[0..n-1]`. We start by assuming that the value we are looking for, called `key`, is somewhere in that range 0 to `n-1`. We can reduce the size of the set that we are searching in by inspecting the value in the middle of the set, namely `X[(n-1)/2]`. If n is odd, this is the middle value. If n is even, we truncate, leading to the data item just below centre.

If `key` is equal to this value, we've found it. If not, then we can eliminate (more than) half the set as a possible location. So, if `key < X[(n-1)/2]`, we know key must be located in the lower half of the set, otherwise it is in the upper half.

Let us suppose that we know the value `key` should be between entry number `L` and entry number `U`. We check entry number `M=(L+U)/2`. If that entry is not `key`, then `key < X[M]` means `key` is in range from `L` to `M-1`, so make `U=M-1`. Otherwise key is in range `L=M+1` to `U`.

How do we stop if `key` is not in the set? We'll at some point set `L` or `U` so that `L > U`.

We know that the method will work (proof of correctness) as we reduce the search set by a factor of 2 each step. Ultimately the size will be 1.

How efficient is it? We need to know how many comparisons will be needed. How many times can we halve a set of n values? That is, for what value of M is $n=2^M$ (approximately)? The answer is $\log_2 n$.

We say that binary search is *O(log n)*. (What base the log is taken to doesn't matter (in discussing order) as we can show $\log_a n = c \log_b n$, and the constant is ignored in order notation. In terms of actual counts, log to base 2 is used.)

Thus we could improve Insertion Sort by using binary search to determine where in the already-sorted set the new entry should go. This would make the number of comparisons *O(nlogn)*. But the number of moves would not change. This indicates that we should not necessarily be swayed by just the number of comparisons.